2

NASA Contractor Report 181900

ICASE Report No. 89-41

ICASE

PARALLEL LANGUAGE CONSTRUCTS FOR TENSOR PRODUCT COMPUTATIONS ON LOOSELY COUPLED ARCHITECTURES

P. Mehrotra
J. Van Rosendale

Contract No. NAS1-18605
September 1989

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Recently, ICASE has begun differentiating between reports with a mathematical or applied science theme and reports whose main emphasis is some aspect of computer science by producing the computer science reports with a yellow cover. The blue cover reports will now emphasize mathematical research. In all other aspects the reports will remain the same; in particular, they will continue to be submitted to the appropriate journals or conferences for formal publication.

Parallel Language Constructs for Tensor Product Computations on Loosely Coupled Architectures*

Piyush Mehrotra[†]
John Van Rosendale

Institute for Computer Applications in Science and Engineering
Hampton, VA 23665.

Abstract

Distributed memory architectures offer high levels of performance and flexibility, but have proven awkward to program. Current languages for nonshared memory architectures provide a relatively low-level programming environment, and are poorly suited to modular programming, and to the construction of libraries. This paper describes a set of language primitives designed to allow the specification of parallel numerical algorithms at a higher level. We focus here on tensor product array computations, a simple but important class of numerical algorithms. We consider first the problem of programming one dimensional "kernel" routines, such as parallel tridiagonal solvers, and after that look at how such parallel kernels can be combined to form parallel tensor product algorithms. [illegible]

*This research was supported by the Institute for Defense Analysis under contract IDA 10-00008, by the Office of Naval Research under contract ONR N00014-88-M-0108, and by the National Aeronautics and Space Administration under NASA contract NAS1-18605 while the authors were in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

[†] On leave from the Department of Computer Science, Purdue University, West Lafayette, IN 47907.

1 Introduction

Distributed memory architectures offer very high levels of performance at modest cost. Machines now becoming available have peak speeds of many gigaflops, at a fraction of the cost of high-end vector multiprocessors, and teraflop machines will be possible in a few years. Given this performance potential, it is clear that such architectures will play an increasing role in scientific computing.

However, there remains a fundamental problem with distributed memory architectures; they tend to be quite awkward to program. Parallel programming is generally harder than sequential programming, since the programmer needs to be aware of the multiple threads of control flow, and the subtle semantic issues which can arise[6]. Nonshared memory architectures compound this difficulty by forcing the programmer to decompose each data structure into separate pieces, each "owned" by one of the processors. Also, on such architectures, all interprocessor communication must be explicitly specified using the low-level message passing constructs supported by the architecture.

Some of the difficulty of programming distributed memory machines seems to be inherent in this class of architectures, though a large fraction appears to be due to a lack of adequate programming tools. The Kali project combines work of the authors with other research on programming and load balancing being carried out at ICASE [2, 12, 18]. The ultimate goal is to ameliorate the problem of programming distributed memory architectures to the extent possible.

Nature of the problem

This paper focuses on the problem of expressing tensor product array computations, for distributed memory architectures. Tensor product algorithms are those in which multidimensional arrays are manipulated by applying operations to lower dimensional slices. Such algorithms offer both simplicity and efficiency. Thus tensor product algorithms are widely used in spline fitting, in picture processing, in computer aided geometry, in computational fluid dynamics, and so forth.

Despite this diversity of applications, the programming issues in tensor product algorithms are universal. One needs effective ways of pealing off lower dimensional slices of arrays, and one needs to be able to apply one of a number of different operations on those slices. In some cases, the algorithm applied to the slices will also be a tensor product algorithm, (c.f. section 5.). When these tensor product algorithms are implemented on distributed memory machines, both the original array, and the operations on the slices may be distributed. This is the issue addressed in this paper.

Finding the proper way of expressing tensor product algorithms on distributed memory architectures is a basic problem. Tensor product algorithms, and array manipulation algorithms more generally, are a natural target for parallel computing research. Virtually all large numerical programs involve array manipulation in some way. Moreover, the regularity of the operations involved, and the potential for highly efficient execution, makes these algorithms especially susceptible to development of effective language constructs and programming techniques.

The plan of this paper is as follows. In section 2, we discuss distributed memory programming, and current work in the the Kali project. Section 3 considers the programming of one dimensional kernel routines, such as tridiagonal solvers. After that sections 4 and 5 look at how these one dimensional kernels can be combined into higher dimensional algorithms for realistic numerical computations. Finally, section 6 briefly discusses the adequacies and limitations of the present approach, and directions in which research appears necessary.

2 Parallel Programming Constructs

Most programming environments for distributed memory architectures are based on "message passing languages." The basic paradigm for such languages is CSP (Communicating Sequential Processes)[7], in which independently executing sequential tasks interact and synchronize through messages. Examples include Occam[20], and message passing dialects of C and Fortran as supplied by machine manufacturers.

In one sense, these message passing languages are ideal, since they accurately embody the set of operations which can be performed by the hardware. However, there are also serious limitations with such languages; the most obvious is the relatively low-level at which algorithms must be specified. The programmer first needs to distribute the data structures across a set of processes, each having its own separate name-space. Then, in order to share values between processes, the programmer must specify a "send" operation in one process, and a matching "receive" operation in another. This kind of programming is more challenging than one would expect, induces a mass of low-level "message passing" detail for even the simplest algorithms, and can easily lead to deadlock or non-determinancy if one is not sufficiently careful[6].

Another problem exists with these languages as well. Modern program design relies heavily on "modular programming" and "top down" design. One builds up large programs as a collection of "modules" or "procedures" each designed to perform a specific subtask. Message passing languages provide no support for this kind of decomposition. One can define "procedures" running on each processor, but there is no notion of a "distributed procedure" running on a collection of processors.

The KALI Project

The Kali project is an attempt to define tools (languages, compilers, performance predictors, etc.) to allow the specification of programs for distributed memory architectures at a higher level, without compromising run-time efficiency. This project is one of several projects addressing the basic issue of high-level programming of distributed memory architectures[1, 21]. To achieve the efficiency and generality of message passing languages, while still permitting high level specification of algorithms is a difficult and multi-faceted problem. In [17] we looked at the problem of programming iterative algorithms utilizing irregular triangular grids, on distributed memory architectures. In this paper, we look at the equally important problem of programming very regular "tensor product" calculations on such architectures.

In this paper, we describe the language constructs we are proposing in a Fortran-like dialect, KF1 (Kali Fortran 1). Our previous papers have generally used a more Pascal like syntax[16, 17]. However, syntax is not the issue. Since most numerical programmers are more comfortable with a Fortran-like syntax, we are adopting this syntax in our current research.

To illustrate the basic concepts in our approach, we consider the simple problem of specifying a Jacobi iteration for a distributed memory architecture. Listing 1 gives a sequential version of a Jacobi algorithm for Poisson's equation on an n by n grid. This is a simple algorithm which can be easily rewritten in a message passing language for parallel execution. A high level view of this kind of code is given in Listing 2.

In the message passing version of the Jacobi code as shown in Listing 2, the algorithm is assumed to be distributed over a p^2 array of processors. The data is divided into $m \times m$ blocks, where $m = n/p$, so that each processor contains a contiguous subarray of the full solution array. Note that the actual array is declared to be $(m+2)\times(m+2)$ so that boundary data can be easily maintained and accessed. Such a distribution of the array data structure keeps most of the array references in the computation local while balancing the load across the processors. The values on the boundaries of the subarray "owned" by each processor need to be communicated to each of its four neighbors at each iteration; hence the sequence of guarded sends and receives shown. The "if" guards are needed, since processors "owning" solution values along physical boundaries of the processor array do not send or receive these values from their neighboring processors. Also, the communication here has been assumed to be asynchronous; if the underlying architecture requires synchronous communications then the sends and receives have to be ordered carefully to avoid deadlock.

The KF1 code for this algorithm, given in Listing 3, looks very much like the sequential Fortran code, though it will compile into message-passing code analogous to that in Listing 2. In addition to the code in the sequential Fortran version, the user must specify three additional kinds of information in KF1:

```
      parameter (np = ...)
      real  X(0:np, 0:np), f(0:np, 0:np)
      real  tmpX(0:np, 0:np)

      n = np - 1
      do 1000 it = 1, 50
c
c        copy solution into a temporary array
c
        do 200 j = 1, n
          do 100 i = 1, n
            tmpX(i, j) = X(i, j)
100         continue
200       continue
c
c        update solution array
c
        do 400 j = 1, n
          do 300 i = 1, n
            X(i, j) = 0.25*(tmpX(i+1, j) + tmpX(i-1, j)
     &                                  + tmpX(i, j+1) + tmpX(i, j-1)) - f(i, j)
300         continue
400       continue

1000    continue
```

Listing 1: Sequential Jacobi algorithm

a) the processor array on which the program is to be executed,

b) the distribution of the data structures across these processors, and

c) the parallel loops specifying the computation on the distributed data structures.

These are aspects of the parallel algorithm that are critical to performance. Automatic generation of such information from sequential code is beyond current compiler technology, and hence it must be supplied by the programmer.

Processor Array

The first thing that needs to be specified is a "processor array." This is an array of physical processors across which the data structures will be distributed, and on which the algorithm will execute. In the *jacobi* routine, the processor array is passed in as an argument.

Code for process P(ip,jp) ...

```
          parameter (mp = ...)
          real  X(0:mp, 0:mp), f(0:mp, 0:mp)
          real  tmpX(0:mp, 0:mp)

          m = mp - 1
          do 1000 it = 1, 50
c
c            copy interior of solution array into a temporary array
c
            do 200 j = 1, m
              do 100 i = 1, m
                tmpX(i, j) = X(i, j)
100           continue
200         continue
c
c         send edge values to North, South, East and West neighbors
c
            if( ip .gt. 1  ) send ( P(ip-1, jp),  X(1, 1:m) )
            if( ip .le. np ) send ( P(ip+1, jp),  X(m, 1:m) )
            if( jp .gt. 1  ) send ( P(ip, jp-1),  X(1:m, 1) )
            if( jp .le. np ) send ( P(ip, jp+1),  X(1:m, m) )
c
c         receive edge values from neighbors
c
            if( jp .le. np )  recv ( P(ip, jp+1),  tmpX(1:m, mp) )
            if( jp .gt. 1  )  recv ( P(ip, jp-1),  tmpX(1:m, 0) )
            if( ip .le. np )  recv ( P(ip+1, jp),  tmpX(mp, 1:m) )
            if( ip .gt. 1  )  recv ( P(ip-1, jp),  tmpX(0, 1:m) )
c
c         update solution array X
c
            do 400 j = 1, m
              do 300 i = 1, m
                X(i, j) = 0.25*(tmpX(i+1, j) + tmpX(i-1, j)
    &                          + tmpX(i, j+1) + tmpX(i, j-1)) - f(i, j)
300           continue
400         continue

1000      continue
```

Listing 2: Message-passing version of Jacobi algorithm

```
      parsub jacobi(X, f, np: procs)

      processors procs(p, p)

      real X(0:np, 0:np), f(0:np, 0:np) dist (block, block)

      n = np - 1
      do 1000 it = 1, 50
c
c        copy solution into a temporary array
c
        doall 100 (i, j) = [1, n] * [1, n] on owner(x(i, j))
              X(i, j) = 0.25*(X(i+1, j) + X(i-1, j) + X(i, j+1) + X(i, j-1)) - f(i, j)
100       continue

1000   continue

      return
      end
```

Listing 3: KF1 version of the Jacobi algorithm

The routine declares the argument *procs* to be a two dimensional processor array, having size *p* by *p*. The size of the processor array argument is "open", and is determined by the actual size of the processor array passed at the point of call. The identifier *p* reflects this size, and can be used in the body of the subroutine as a constant. The keyword **parsub** declares *jacobi* to be a parallel subroutine, which will execute in parallel on distributed data. A processor argument can be passed only to parallel subroutines and each parallel subroutine must either declare a processor array or must be passed one as an argument.

Only one "real" processor declaration is allowed in the whole program, and it generally occurs in the main program. The processor array (or its slices) can then be passed around as a parameter for parallel execution of subroutines. The initial processor declaration is the "real estate agent" discussed by C. Seitz. The semantics and behavior of the real estate agent is interesting, but is tangential to the issues here.

Data Distribution Primitives

Given a processor array, the programmer must specify the distribution of data structures across that processor array. In the current version of KF1, the only distributed data type supported is distributed arrays. Array distributions are specified by a *distribution clause* in their declaration. This clause specifies a sequence of distribution patterns, one for each dimension of the array. Scalar variables and arrays without a distribution clause are simply replicated with one copy assigned to each of the processors in the processor array.

Each dimension of a data array can be distributed across the processors in one of several patterns, or can be left undistributed. In this subroutine, the data array X is distributed with a (**block**, **block**) distribution. This means that each dimension is "blocked," so that each processor receives a square subarray of the full array X. Another kind of distribution is a **cyclic** distribution, especially useful in numerical linear algebra, in which the elements are distributed in a round-robin fashion across the processors. The number of dimensions of an array that are distributed must match the number of dimensions of the underlying processor array. Asterisks are used to indicate dimensions of data arrays which are not distributed.

When distributed data structures are passed as arguments to parallel subroutines, the set of processors owning the portions of the data structures being passed, needs to be passed as well. Thus, for example, passing a slice of a distributed array often entails passing a matching slice of the processor array. This idea should become clearer through the examples here.

Doall Loops

Operations on distributed data structures are specified by **doall** loops. The **doall** loop here is similar to the **forall** loop in BLAZE[16], and to parallel loops in other parallel dialects of Fortran. The example below shows a loop which performs $n-1$ loop invocations, shifting the values in the array A one space to the left.

```
        doall 100 i = 1, n-1  on  owner(A(i))
            . . .
            A(i) = A(i+1)
            . . .
100     continue
```

The semantics here are "copy-in copy-out," in the sense that the values on the right hand side of the assignment are the old values in array A, before being modified by the loop. Thus the array A is effectively, "copied into" each invocation of the **doall** loop, and then the changes are "copied out." Thus no temporary array is required in the *jacobi* routine given in Listing 3.

In addition to the range specification in the header of the **doall**, there is also an **on** clause. This clause specifies the processor on which each loop invocation is to be executed. In the above program fragment, the **on** clause causes the ith loop invocation to be executed on the processor owning the ith element of the array A.

The **on** clause associated with a **doall** loop allows the compiler to partition the loop invocations among the processors participating in the loop. This process, called "strip-mining", is fairly simple given all the information available to the compiler [12, 13]. Note that the code outside the **doall** loops is replicated in each of the processors.

The loop headers of purely nested **doall** loops can be combined into a single header as shown in Listing 3 for the two loops of the *jacobi* routine. Here, a product of the ranges is used to specify that for each value of the outer loop variable i, in the range $[1,\ n]$, the inner loop variable j assumes each of the values in the range $[1,\ n]$.

Specifying Communication

Using KF1, a programmer can specify a data parallel algorithm at a high level, while still retaining control over those details critical to performance. The additional information required of the user here is exactly the information most critical to performance. Note that the body of the **doall** loop here is independent of the distribution of the array X and of the processor array P. Thus a variety of distribution patterns can be tried by simple modifications of this program. This makes "tuning" of parallel programs much easier with this kind of language than it is with message passing languages.

It is important to note that KF1 contains no explicit communication constructs. The programmer specifies distribution of data values across the processors, and also specifies the location where operations are to be performed. From this, the compiler produces the low-level details of the message passing code to be executed on the architecture by a sequence of program transformations [12, 17].

In general, given an assignment statement, like that in the Jacobi example, the compiler can decide which processor "owns" each of the values on the left and right hand sides of the equation, and can then generate efficient message passing code. This is true of all of the examples in this paper, and of most others we have studied. In other cases, the compiler must generate runtime code which will gather such information on the fly[17].

However, this issue of how communication is expressed is subtle; it is not clear which approach is best. The choice in KF1, of leaving communication implicit, appears natural from the users point of view, since it dramatically simplifies programming. It also greatly simplifies "tuning" of parallel programs, and allows a "modular" or "top down" design strategy which is impossible with Occam-style languages. However, there is a serious defect here: benign looking code will sometimes run exceptionally slowly. This is because the compiler cannot always generate effective message passing executable code, particularly for complicated loops.

We plan to address this issue by providing performance estimation tools, which will indicate which parts of a program will compile into efficient executable code, and which will not. Given such a tool, a programmer should have little trouble designing efficient programs. In the end, however, the effectiveness of this kind of language will depend on many factors, including the communications capabilities of architectures, the quality of compilers, and the needs of working programmers. Resolving these issues is an area of active research.

3 Parallel Tridiagonal Solvers

In this section, we describe the programming of a parallel algorithm for tridiagonal systems of equations, using the KF1 primitives of the last section. Solving tridiagonal systems is a common "kernel algorithm" for multi-dimensional tensor product algorithms. Other "one-dimensional kernels" frequently needed are cubic spline fitting routines, Fast Fourier Transforms, and so forth, but tridiagonal solvers are the most commonly used. From a programming point of view, all of these kernel algorithms are similar, and most can be treated by analogous "divide and conquer" techniques on parallel architectures.

Consider the problem of solving a tridiagonal matrix of equations of size n. Let A be the tridiagonal matrix whose ith row has nonzero elements $(b_i,\ a_i,\ c_i)$, as shown in Figure 1. We seek the solution X of the tridiagonal system,

$$A\,X = f$$

assuming that the matrix A can be factored without pivoting.

There are a wide variety of parallel tridiagonal algorithms in the literature[8]. The particular one described here is a "substructured" algorithm, which is a variant of Sameh's "spike" algorithm[5]. This algorithm is a tree-structured "divide and conquer" algorithm executing on p processors. In the first phase of the algorithm, we perform a sequence of $log_2(p)$ reduction steps, each halving the size of the tridiagonal system being solved. In the second phase of the algorithm, we perform substitution to obtain the solution.

The matrix A is assumed to be distributed by blocks of rows across the p processors. Given this distribution of the array, processor i is responsible for rows $l_i = (i-1)n/p + 1$ through $u_i = in/p$. In the first step, processor i performs elimination on rows $l_i + 2$ through u_i, eliminating the lower diagonal of the tridiagonal system, but introducing fill-in in column l_i. Next, it performs elimination in the reverse direction on rows $u_i - 2$ through l_i, eliminating the upper diagonal, while introducing fill-in in column u_i. Thus at the end of this step, the upper and lower diagonals are eliminated in rows $l_i + 1$ through $u_i - 1$. Moreover, rows l_i and u_i are now coupled directly to each other, and contain no entries corresponding to the intermediate rows Thus rows $l_1, u_1, l_2, u_2, ..., l_p, u_p$ now constitute a tridiagonal system having $2p$ equations, as is shown by the highlighting in Figure 1.

In the second step, the pair of equations corresponding to rows l_i and u_i on each processor are "mailed" to some processor. Half of the processors "receive" two pairs of equations, constituting four adjacent rows of the matrix, and remain "active." The other half of the processors receive no equations, and go to sleep. Thus one active processor will receive rows l_1, u_1, l_2, u_2, another rows l_3, u_3, l_4, u_4, and so on.

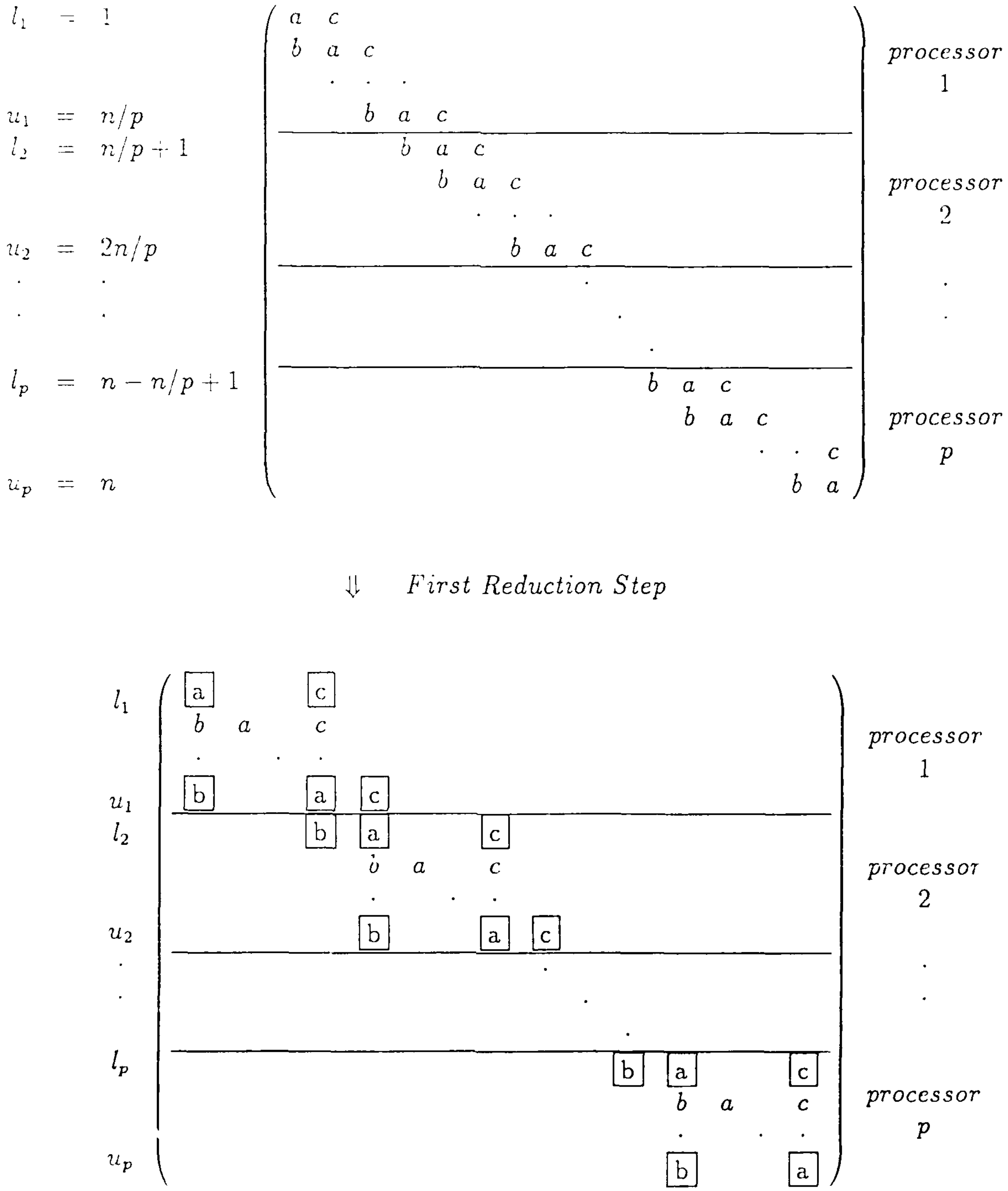

Figure 1: Reduction in the first step of the elimination process.

$$\begin{array}{c|cccc|c} b & a & c & & & \\ & b & a & c & & \\ & & b & a & c & \\ & & & b & a & c \end{array} \Rightarrow \begin{array}{c|cccc|c} \boxed{b} & \boxed{a} & & & \boxed{c} & \\ & b & a & & c & \\ & b & & a & c & \\ & \boxed{b} & & & \boxed{a} & \boxed{c} \end{array}$$

Figure 2: Reduction of four rows of a tridiagonal system.

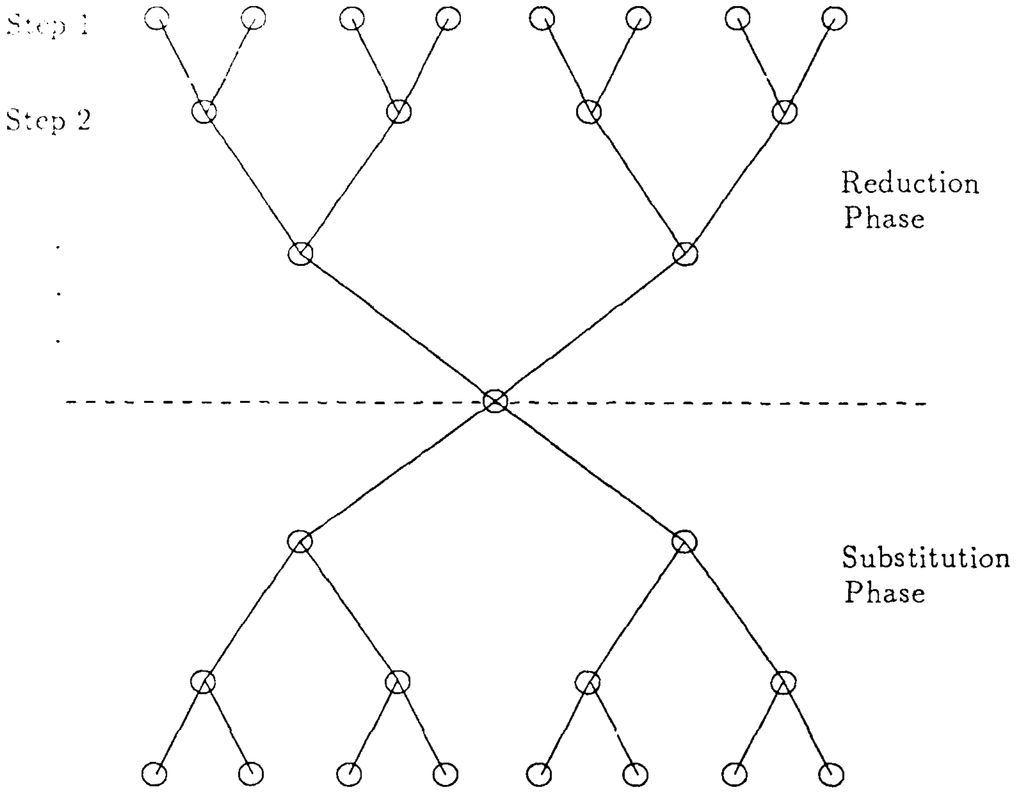

Figure 3: Data flow graph.

These four equations on each active processor are then reduced to two, as shown in Figure 2, just as in the first step, so that the first and last equations on each active processor are directly coupled. The result is a tridiagonal system of size p. This process continues, halving the size of the tridiagonal system at each step in the reduction. After $log_2(p)$ such steps, we obtain a single tridiagonal system having four rows, which we solve by the sequential Thomas algorithm.

During the $log_2(p)$ steps of this reduction phase, each of the reduced linear systems occurring must be saved. Then after the final tridiagonal system with four equations has been solved, we perform substitution into these saved reduced systems, in the inverse order

$$\begin{pmatrix} - & & & - \\ b & a & & c \\ b & & a & c \\ - & & & - \end{pmatrix}\begin{pmatrix} y \\ x_1 \\ x_2 \\ y \end{pmatrix} = \begin{pmatrix} - \\ f \\ f \\ - \end{pmatrix}$$

Figure 4: Computation of intermediate values x_1 and x_2

in which they were created, and finally recover the solution of the original system. The overall data flow graph of this is substructured algorithm is shown in Figure 3.

The substitution process itself is quite trivial. At each step, each of the active processors must compute its share of the solution of a reduced system. Each processor receives the first and last values of the solution of this part of the reduced system, as shown in Figure 4, and computes the intermediate values, as shown. In the first $log_2(p) - 1$ steps of the substitution phase, two intermediate solution values need to be computed, and then these 4 solution values are mailed to the processors needing them in the next step. In the last step, each processor computes $n/p - 2$ solution values, completing the solution.

Mapping of data flow graph unto processor array

The data flow graph of this substructured algorithm is shown in Figure 3. During the reduction phase, the number of active processors is reduced by two at each step, until finally we have just one active processor. During the substitution phase, the number of active processors doubles at each stage.

There are various ways of mapping this data flow graph onto a multiprocessor architecture. One of the simplest is the shuffle/unshuffle mapping shown in Figure 5. This mapping is easy to program, and is advantageous when there are multiple tridiagonal system to be solved, as we will show later.

KF1 representation of algorithm

It is relatively easy to describe this kind of divide and conquer algorithm in KF1. Listing 4 gives the KF1 code for this algorithm. Subroutine *tri* takes as input vectors b, a, and c representing the matrix A and the right hand side f, and returns the solution vector X. These vectors are declared to be distributed by blocks across a one-dimensional processor array *procs* of size p. Note that the routine uses the new declaration **dynamic** for temporary arrays, e.g. *tmpa*, whose sizes can be determined only when the subroutine is invoked. Fortran programmers generally perform dynamic allocation "by hand," by indexing into the blank common area. Such techniques are awkward and difficult to handle on distributed

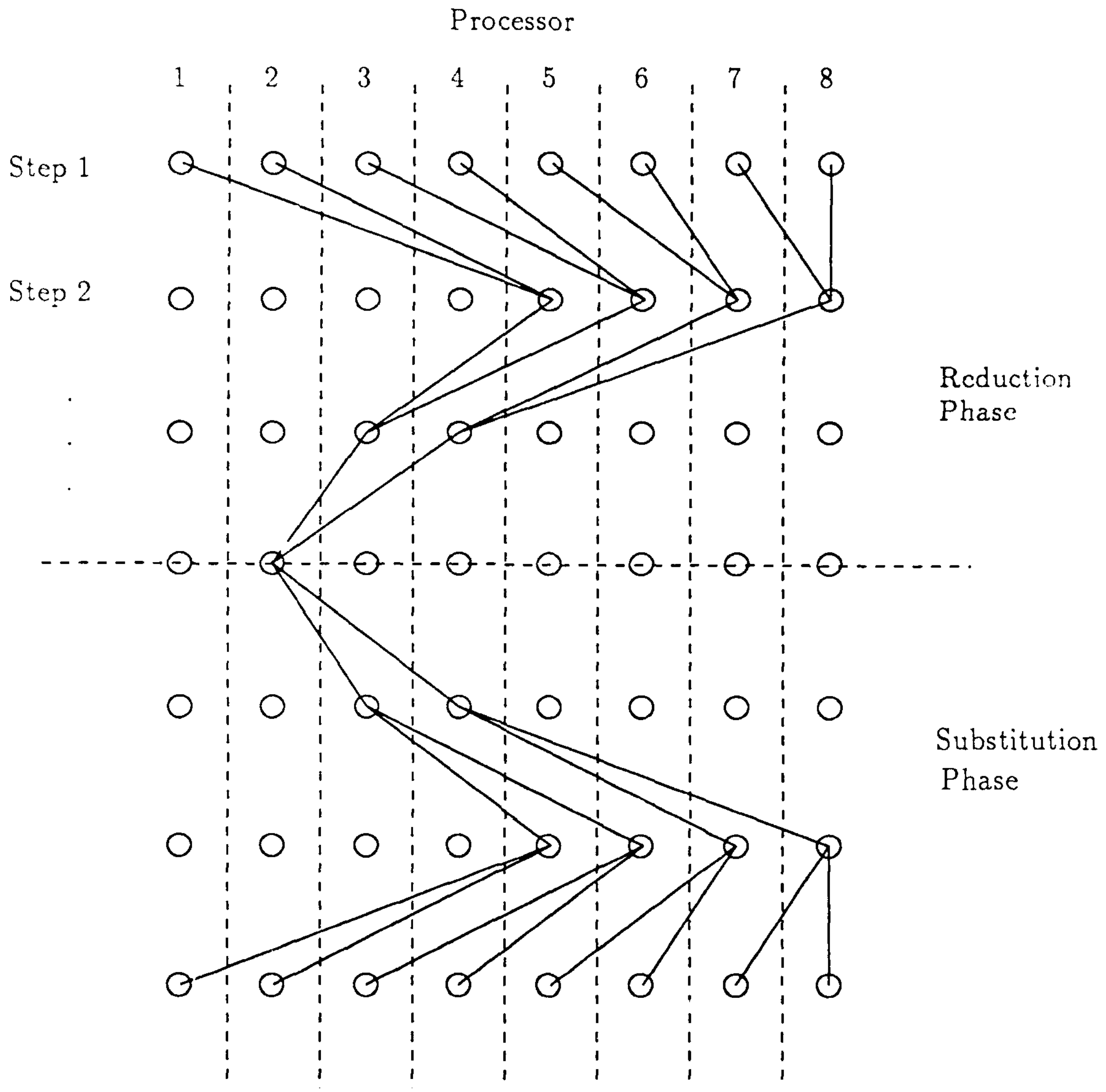

Figure 5: Mapping of data flow graph.

memory machines, hence "dynamic arrays" have been included in KF1.

The outer **do** loop here executes the $log_2(p)$ steps of the algorithm. In each step, first the internal equations are eliminated using a **doall** loop and then the outer two equations are sent to another processor. In the first step, the number of internal equations to be eliminated depends on the original distribution of the rows of the matrix, while in the later steps each processor reduces a system of four equations. The first **doall** loop is executed only in the first step. The **on** clause "**on** *procs(ip)*" specifies that the *ip*th invocation of the loop is to be executed on processor *procs(ip)*. Each processor uses the predefined functions **lower** and **upper**, to determine the index limits for the block of equations that it "owns". A sequential

```
      parsub tri( X, f, b, a, c, n; procs)

      processors procs(p)
      real  X(n), f(n) dist(block)
      real  a(n), b(n), c(n) dist(block)
      integer lo, hi, step

      dynamic real tmpa(4*p), tmpb(4*p) dist(block)
      dynamic real tmpc(4*p), tmpf(4*p) dist(block)

      k = log2(p)
      do 1000 step = 1, k
        if (step .eq. 1) then
              doall 100 ip = 1, p on procs(ip)
                lo = lower (X, procs(ip) )
                hi = upper (X, procs(ip) )
                call reduce(b(lo:hi), a(lo:hi), c(lo:hi), f(lo:hi), hi-lo+1)
                tmpb(4*ip-3) = b(lo)
                tmpb(4*ip)   = b(hi)
c
c               ... code to set up tmpa, tmpc, tmpf similarly
c
100           continue
        else
              doall 200 ip = 1, p on procs(ip)
                 if( log2(ip)+step .eq. k+1 ) then
                    lo = 4*ip - 3
                    hi = 4*ip
                    call reduce(tmpb(lo:hi), tmpa(lo:hi), tmpc(lo:hi), tmpf(lo:hi), 4)
                 endif
200           continue
        endif

        call unshff(tmpb, tmpa, tmpc, tmpf, 4*p, step; procs)

1000  continue
c
c       ... code for substitution phase
c
      return
      end
```

Listing 4: Code for a tridiagonal solve

routine *reduce* is then called to eliminate the internal equations of the block of equations owned by the processor. Routine *reduce* is a simple sequential linear algebra routine, not shown. After elimination of the internal equations in the first step, the outer two equations are transferred to temporary arrays to be communicated to the next step of the elimination process.

In each step, the routine *unshff* (shown in Listing 5) is called to permute the equations among the processors as needed. Given the simple distribution pattern here, the compiler can convert the assignment statements representing the permutation in *unshff*, into sends and receives required for communicating the data[1].

After the equations have been moved, the four equations in each of the active processors are again reduced to two by calling the routine *reduce*. This is done in the second **doall** loop, where the "if" condition controls which processors are active.

Pipelined parallel tridiagonal algorithm

In the above tridiagonal algorithm, the number of active processors is halved at each phase. If we have to solve more than one tridiagonal system then these computations can be pipelined so that more of the processors are kept busy. Listing 6 shows a pipelined version of the tridiagonal solver. The subroutine *mtrix* accepts m tridiagonal systems each of size n represented by $m \times n$ arrays. Note that the second dimension of the data arrays here is distributed across the one dimensional processor array, i.e., each processor contains a block of each of the m tridiagonal systems.

Here, in each of the first m steps, a new set of equations are reduced to yield a set of $2p$ equations. The second **doall** loop is setup such that different subsets of the processors "handle" different sets of equations during a step. The routine *reduce* is the same as discussed before while routine *munshf* is similar to *unshff* except that it needs to know which particular set of equations are to be "unshuffled".

4 Two Dimensional Tensor Product Computations

In the last section we presented simple one dimensional kernel algorithms in KF1. This section shows how such one dimensional kernels can be combined for two dimensional tensor product calculations. The example chosen here is an ADI iteration. Mapping ADI methods to distributed memory architectures has been previously studied by several groups[9, 14].

ADI (Alternating Direction Implicit) is a well known and effective method for solving

[1] The transformation here would be trivial, if the user gave a pragma specifying "inline" expansion of procedure *unshff*. Without such a pragma, the inter-procedural analysis required is difficult, and is the focus of ongoing research.

```
      parsub unshff( b, a, c, f, n, phase; procs)

      processors procs(p)
      real  b(n), a(n), c(n), f(n) dist (block)
      integer phase, dest, src

      k = log2(p)
      doall 100 ip = 1, p on procs(ip)
         if( log2(ip)+phase .eq. k+1 ) then
               dest = lower (b, procs)
               src =  (2*lo) % n

               b(dest)   = b(src-1)
               b(dest+1) = b(src+2)
               b(dest+2) = b(src+3)
               b(dest+3) = b(src+6)
c
c     ...   similarly for a, c, and f
c
         endif
100   continue

      return
      end
```

Listing 5: Routine to "unshuffle" equations.

partial differential equations in two or more dimensions[15, 19]. It is widely used in computational fluid dynamics, and other areas of computational physics. The name ADI derives from the fact that "implicit" equations are solved in both the x and y directions at each step. These implicit equations are often tridiagonal systems, and can be solved by parallel tridiagonal solvers, such as those described in the last section. Thus the main task here is to show how calls to these parallel tridiagonal routines are combined, to specify the ADI algorithm. Almost any working numerical analyst would know how one does that in Fortran. With a language like KF1, the same approach yields a distributed parallel implementation.

Mathematically, ADI works as follows. Suppose one is trying to solve a partial differential equation:

$$a(x,y)U_{xx} + b(x,y)U_{yy} + c(x,y)U = f$$

Viewing this as an operator equation

$$LU = f, \tag{1}$$

```
      parsub mtrix( X, f, b, a, c, m, n; procs)

      processors procs(p)
      real  X(m, n), f(m, n) dist(*, block)
      real  a(m, n), b(m, n), c(m, n) dist(*, block)
      integer lo, hi, step

      dynamic real tmpa(m, 4*p), tmpb(m, 4*p) dist(*, block)
      dynamic real tmpc(m, 4*p), tmpf(m, 4*p) dist(*, block)

      k = log2(p)
      do 1000 step = 1, m+k
        if (step .le. m) then
            doall ip = 1, p on procs(ip)
              lo = lower (X(step, *), procs(ip) )
              hi = upper (X(step, *), procs(ip) )
              call reduce(b(step, lo:hi), a(step, lo:hi),
   &                        c(step, lo:hi), f(step, lo:hi), hi-lo+1)
              tmpb(step, 4*ip-3) = b(step, lo)
              tmpb(step, 4*ip)   = b(step, hi)
c
c             ... code to set up tmpa, tmpc, tmpf
c
100         continue
        else
            doall ip = 1, p on procs(ip)
              j = step + log2(ip) - k
              if (j .gt. 1 .and. j .le. m) then
                  lo = 4*ip - 3
                  hi = 4*ip
                  call reduce(tmpb(j, lo:hi), tmpa(j, lo:hi),
   &                            tmpc(j, lo:hi), tmpf(j, lo:hi), 4)
              endif
200         continue
        endif

      call munshf(tmpb, tmpa, tmpc, tmpf, 4*p, step; procs)

1000  continue
c
c     ... code for substitution phase
c
      return
      end
```

Listing 6: Pipelined version to solve m tridiagonal systems

one decomposes L into a sum of two parts:

$$L_1 U + L_2 U = f,$$

where

$$\begin{aligned} L_1 &= a(x,y)\frac{\partial}{\partial xx} + c(x,y)/2 \\ L_2 &= b(x,y)\frac{\partial}{\partial yy} + c(x,y)/2 \end{aligned}$$

Then instead of directly solving equation 1, one successively solves

$$(L_1 + I)v = f \tag{2}$$

and

$$(L_2 + I)u = v. \tag{3}$$

Carrying out these two operations gives a first approximation to the solution of equation 1. After this one replaces the right hand side f by the residual

$$r = Lu - f, \tag{4}$$

and repeats the process. Continuing, one has an efficient iterative method, which converges to the solution of equation 1.

The advantage of this algorithm over competing iterative methods is that it converges quite rapidly, and the solutions to the equations 2 and 3 only require inexpensive tridiagonal solves. Listing 7 presents this algorithm in KF1. This version of ADI uses the non-pipelined parallel tridiagonal solver. Here *resid* is a simple subroutine which forms the residual of equation 4. This residual computation is similar to one step of a Jacobi iteration, and induces the same communication.

The **on** clauses here force each loop invocation to be performed on the appropriate slice of the two-dimensional processor array. The construct “**owner**((r(i,*))” specifies the set of processors which own the *i*th row of the array r. The *i*th tridiagonal system in the y-direction is solved by calling the subroutine, *tric*, as follows:

call tric(v(i, *), r(i, *), b0, b1, b0, ny; **owner**(r(i, *)))

The routine is passed a slice of the data arrays v and r, along with the slice of the processor array on which these values reside so that it can execute in parallel. The tridiagonal solver, *tric*, here is just the constant coefficient version of routine *tri* presented in section 3. Programming ADI with variable coefficients is not much different, except that there are a number of additional details not germane to this paper.

```
      parsub adi (u, f, nxp, nyp; procs)

      processors procs(px, py)

      real  u(0:nxp, 0:nyp), f(0:nxp, 0:nyp) dist (block, block)
      dynamic real  r(0:nxp, 0:nyp), v(0:nxp, 0:nyp) dist (block, block)

      common /params/ maxits, a, b, c
c
c     ADI iteration for the constant coefficient problem
c           a*Uxx + b*Uyy + c*U = F
c
      nx = nxp-1
      ny = nyp-1

      a0 = a/(nx*nx)
      a1 = c - 2*a0
      b0 = b/(ny*ny)
      b1 = c - 2*b0

      do 1000 it = 1, maxits

         call resid(r, u, f, nx, ny; procs)
c
c        perform tridiagonal solves in y direction
c
         doall 100 i = 1, nx on owner(r(i, *))
               call tric(v(i, *), r(i, *), b0, b1, b0, ny; owner(r(i, *)) )
100      continue
c
c        perform tridiagonal solves in x direction
c
         doall 200 j = 1, ny on owner(v(*, j))
               call tric(u(*, j), v(*, j), a0, a1, a0, nx; owner(v(*, j) )
200      continue

1000  continue

      return
      end
```

Listing 7: ADI Algorithm

```
        parsub madi (u, f, nxp, nyp; procs)

        processors procs(px, py)

        real  u(0:nxp, 0:nyp), f(0:nxp, 0:nyp) dist (block, block)
        dynamic real  r(0:nxp, 0:nyp), v(0:nxp, 0:nyp) dist (block, block)
        integer lo, hi

        common /params/ maxits, a, b, c

        nx = nxp-1
        ny = nyp-1

        a0 = a/(nx*nx)
        a1 = c/2 + 2*a0
        b0 = b/(ny*ny)
        b1 = c/2 + 2*b0

        do 1000 it = 1, maxits
           call resid(r, u, f, nx, ny; procs)

c          perform tridiagonal solves in y direction

           doall 100 ip = 1, px on procs(ip, *)
                 lo = lower(v, procs(ip, *), 1)
                 hi = upper(v, procs(ip, *), 1)
                 call mtrixc(v(lo:hi, *), r(lo:hi, *), b0, b1, b0, hi-lo+1, ny; procs(ip, *))
100        continue

c          perform tridiagonal solves in x direction

           doall 200 jp = 1, py on procs(*, jp)
                 lo = lower(v, procs(*, jp), 2)
                 hi = upper(v, procs(*, jp), 2)
                 call mtriyc(u(*, lo:hi), v(*, lo:hi), a0, a1, a0, nx, hi-lo+1; procs(*, jp))
200        continue
1000    continue

        return
        end
```

Listing 8: Pipelined ADI Algorithm

Use of the non-pipelined tridiagonal solver here is somewhat inefficient, since each processor shares in the solution of *nx/mx* tridiagonal systems during the y-direction solution, and *ny/my* tridiagonal systems during the x-direction solutions. One can get better speedups with the pipelined version of the tridiagonal solver. The improved algorithm is given in Listing 8. Again, the parallel tridiagonal solver, *mtrixc*, used here is the constant coefficient version of subroutine *mtrix* of section 3. Separate routines *mtrixc* and *mtriyc* are needed, since the arrays passed as arguments will be transposed during the y-direction part of the ADI iteration

One can think of the difference between this second pipelined version of ADI, and the first as being a difference in the extent to which one leaves "strip-mining" of parallel loops to the compiler. In principal, a sufficiently good compiler could generate this second version of ADI from the first, simply by code restructuring. This is rather difficult however, since the compiler would first have to merge multiple calls to subroutine *tri*, and then reschedule the operations in the tridiagonal solver in complicated ways. This is well beyond the capabilities of existing compilers; for the present, programmers wishing improved performance will have to perform such transformations by hand.

5 Higher Dimensional Tensor Product Algorithms

The last section looked at two dimensional tensor product computations. One could plausibly program such algorithms in Occam-like languages, though it would be awkward. However, that ceases to be the case when one looks at more complex tensor product algorithms. In this section we look at a three dimensional multigrid algorithm having complexity approaching that of real applications. In our view, having a "tool" like KF1 would be a virtual necessity for routine programming of algorithms like this. The idea of designing and debugging such algorithms in an Occam-like language is rather daunting.

Tensor product algorithms are widely used for three dimensional numerical computations. This section examines the expression of a three dimensional multigrid algorithm based on "zebra" relaxation. This algorithm is moderately complex, and is interesting since the plane solves required in the zebra relaxation are themselves tensor product multigrid algorithms.

The multigrid algorithm here is one of a number of related multigrid algorithms based on plane relaxation. This particular one uses "semi-coarsening" and plane relaxations in only one direction[3, 4]. We mention these numerical details only to make it clear that this algorithm is not artificially complex. Quite the opposite in fact; algorithms of much greater complexity are routinely used for modeling of physical problems.

Listing 9 presents the subroutine *mg3* which performs one multigrid iteration on a three dimensional rectangular grid. We assume the partial differential equation being solved is a simple Poisson-like equation, with constant coefficients, and also assume homogeneous Dirichlet boundary conditions.

```
      parsub mg3(u, f, nx, ny, nz; procs)

      processors procs(px, py)

      real u(0:nx, 0:ny, 0:nz), f(0:nx, 0:ny, 0:nz) dist (*, block, block)
      dynamic real r(0:nx, 0:ny, 0:nz) dist (*, block, block)
      dynamic real v(0:nx, 0:ny, 0:nz/2), v(0:nx, 0:ny, 0:nz/2) dist (*, block, block)


c
c     perform zebra relaxation on even planes
c
      call resid3(r, u, f; procs)
      doall 100 k = 2, nz-2, 2 on owner(u(*, *, k))
         call mg2(u(*, *, k), r(*, *, k); owner(u(*, *, k)))
100   continue
c
c     perform zebra relaxation on odd planes
c
      call resid3(r, u, f; procs)
      doall 200 k = 1, nz-1, 2 on owner(u(*, *, k))
         call mg2(u(*, *, k), r(*, *, k); owner(u(*, *, k)))
200   continue
c
c     recursively solve coarse grid problem
c
      if (nz .gt. 2) then
         call resid3(r, u, f; procs)
         call rest3(g, r; procs)

         call mg3(v, g; procs)
         call intrp3(u, v; procs)
      endif

      return
      end
```

Listing 9: Three Dimensional Multigrid Solver

There are two basic operations here, zebra relaxation and solution of coarse grid problems. The zebra relaxations are given by the two **doall** loops. The first performs half of a zebra sweep by visiting the odd planes, while the second visits the even planes, to complete the zebra relaxation. The calls to *resid3* before each **doall** loop compute the "residual," that is, the amount by which we currently fail to satisfy the differential equation. The relaxation

```
      parsub intrp3(u, v, nx, ny, nzf, nzc; procs)

      processors procs(px, py)

      real u(0:nx, 0:ny, 0:nzf), v(0:nx, 0:ny, 0:nzc) dist (*, block, block)
c
c     check whether coarse and fine grid dimensions are properly related
c
      if (nzf .ne. 2*nzc) call error("Dimensions do not match in intrp3")

      do 300 i = 1, nx-1
c
c        modify values on even planes ...
c
         doall 100 (j, k) = [1, ny-1] * [2, nzf-2, 2] on owner(u(*, j, k))
            u(i, j, k) = u(i, j, k) + v(i, j, k/2)
100      continue
c
c        modify values on odd planes ...
c
         doall 200 (j, k) = [1, ny-1] * [1, nzf-1, 2] on owner(u(*, j, k))
            kp = (k+1)/2
            km = (k-1)/2
            u(i, j, k) = u(i, j, k) + 0.5*(v(i, j, kp) + v(i, j, km))
200      continue

300   continue

      return
      end
```

Listing 10: Three Dimensional Interpolation Routine

itself is performed by the calls to *mg2*, each of which "solves" a two dimensional partial differential equation for the solution values on one plane.

The heart of the multigrid algorithm is the "coarse grid correction" in which we recursively call subroutine *mg3* with a smaller problem to accelerate the convergence. In the algorithm here, subroutine *mg3* is called with arrays v and g, which are half as large as the original arrays u and f, since the number of points in the z-direction, nz, is halved.

Postponing temporarily issues related to parallelism, lets look briefly at the various kernel subroutines. Subroutine *intrp3* is typical. One possible variant of this routine is given in Listing 10. This subroutine modifies the current fine grid solution u, using the value of the coarse grid correction v. Since v exists only on a coarse grid having half as many points, values at intermediate points must be computed by interpolation. The simple linear interpolation formula

$$0.5 * (v(i,j,kp) + v(i,j,km)),$$

giving the intermediate value as the average of the two nearest values, is used here. Note that more efficient execution would probably be achieved if the sequential i loop was nested inside the doall loops. This is a trivial program transformation which a good compiler should be able to perform.

Subroutines *resid3* and *rest3* are analogous to *intrp3*, except the numerical formulas occurring are more involved. Subroutine *mg2*, shown in Listing 11, is the two dimensional analog of *mg3*.

The reader interested in numerical details is referred to [3, 4, 10, 22]. Our primary interest here is in the expressiveness of the language constructs. Readers familiar with the programming of such algorithms in sequential Fortran should be able to see that there is very little difference between the programming of such algorithms in Fortran and in KF1.

The other interesting issue here is that of parallelism. The heart of this issue is the calls to *mg2* in the zebra relaxations:

```
c
c   perform zebra relaxation on even planes
c
          call resid3(r, u, f; procs)
          doall 100 k = 2, nz-2, 2  on owner(u(*, *, k))
                call mg2(u(*,*,k), r(*,*,k); owner(u(*, *, k)))
100       continue
```

Given our initial blocking of the three dimensional arrays

```
          processors procs(px, py)
          real  u(0:nx, 0:ny, 0:nz), f(0:nx, 0:ny, 0:nz)  dist (*, block, block)
```

subroutine *mg2* is passed a slice of the processor arrays corresponding to the x-y planes:

```
u(*, *, k) and r(*, *, k)
```

Thus subroutine *mg2* inherits a one dimensional processor array, while its kernel tridiagonal solver, *seqtri*, runs sequentially.

```
      parsub mg2(u, f, nx, ny; procs)

      processors procs(px, py)

      real  u(0:nx, 0:ny), f(0:nx, 0:ny) dist (*, block)

      dynamic real  r(0:nx, 0:ny) dist (*, block)
      dynamic real  v(0:nx, 0:ny/2), v(0:nx, 0:ny/2) dist (*, block)
c
c     perform zebra relaxation on even planes
c
      call resid2(r, u, f; procs)
      doall 100 j = 2, ny-2, 2 on owner( u(*, j) )
         call seqtri(u(*, j), r(*, j))
100   continue
c
c     perform zebra relaxation on odd planes
c
      call resid2(r, u, f; procs)
      doall 200 j = 1, ny-1, 2 on owner( u(*, j) )
         call mg2(u(*, j), r(*, j))
200   continue
c
c     recursively solve coarse grid problem
c
      if(ny .gt. 2) then
         call resid2(r, u, f; procs)
         call rest2(g, r; procs)

         call mg2(v, g; procs)
         call intrp3(u, v; procs)
      endif

      return
      end
```

Listing 11: Two Dimensional Multigrid Solver

We could have done things differently by changing the dimensionality of the original processor array in *mg3*. Had we used a three dimensional processor array there, the tridiagonal solves in *mg2* would have been parallel. Conversely, if we had used a one dimensional processor array, with the three dimensional arrays distributed

real u(0:nx,0:ny,0:nz), f(0:nx,0:ny,0:nz) **dist** (*,*, **block**) subroutine mg2 would have been sequential.

The best alternative here depends on the problem size, the number of processors in the architecture, the cost of communication, and so on. Such issues are outside the scope of this paper. However, it is worth noting that with KF1-like languages, one can experiment with alternate array distributions very easily. It is only necessary to change the blocking patterns in the array declarations, and change some of the *doall* to *do* loops, or conversely. This is a fundamental difference between this kind of language, and Occam-style languages. With message passing languages, the structure of the array distributions will be embedded throughout the program, and thus very difficult to change.

6 Conclusions

As distributed memory architectures change from awkward research curiousities, to production machines for real applications, more attention needs to be focused on programming environments. Experience has shown that message passing languages suffice, but are extremely awkward for complex algorithms such as those described here. The message passing version of a program is often five to ten times longer than the sequential version. In addition the intricate "message plumbing" makes programs difficult to debug, and "hard wires" all algorithm choices, preventing easy experimentation with alternate algorithm designs.

One fundamental problem with such message passing languages is that they provide no support for "distributed procedures." On top of this, the massive amounts of low level message passing detail required to express complex algorithms is a severe impediment. Languages like KF1 and those in related projects, [1, 21], appear to provide a better alternative.

Most of this paper was focused on the issue of the expressiveness of languages like KF1, for multi-dimensional tensor product algorithms. Tensor product algorithms are extremely common. They are the basis of most numerical weather prediction programs, and are heavily used in computational fluid dynamics, computer aided geometric design, and so forth. These algorithms are simple and efficient, and tend to be the method of choice on problems where they are applicable

As we have shown, it is relatively easy to describe tensor product algorithms in KF1. Moreover, the executable code generated will be efficient, since the programmer retains control over data distribution and scheduling. In the examples here, there would be no difference between the execution time of algorithms expressed in KF1, and those expressed in a message passing language, assuming equally good back-end machine code generators. The price of using KF1 instead of a message-passing language is simply slower compilations, since there are additional compiler transformations to be performed.

However, despite the ubiquity and importance of tensor product algorithms, it seems clear that the usability and generality of programming constructs for distributed memory architectures will be determined largely by their success on more complex problems, such as those involving adaptive or irregular grids and general sparse matrices. We are addressing these issues in the Kali project as well. In this paper we have focused on the simpler problem of expressing tensor product algorithms, since this topic raises a concise set of issues, and has intrinsic importance. We also believe *that many of the ideas here apply as well to more* complex problems. Thus the study of tensor product algorithms can be thought of as a valuable jumping-off-point for design of language constructs for harder problems.

A compiler for KF1 is being implemented, and a compiler for a related Pascal-like language has been implemented by C. Koelbel at Purdue University as part of his thesis research [11]. We plan to compare the run-time performance and expressivity of KF1 with that of related languages for distributed memory machines in the coming months. However, raw performance is clearly not the real issue; *the deeper issue is* "usability." *How* do users wish to express algorithms, what level of control do they need/want, which kinds of constructs seem natural. In the end, it is up to the user community to resolve these issues.

References

[1] D. Callahan and K. Kennedy. Compiling programs for distributed memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[2] K. Crowley, J.H. Saltz, R. Mirchandaney, and H. Berryman. Runtime scheduling and execution of loops on message passing machines. Technical Report 89-7, ICASE, January 1989.

[3] N. Decker and J. Van Rosendale. Operator induced multigrid algorithms using semirefinement. Technical Report 89-24, ICASE, 1989.

[4] J.E. Dendy, S. McCormick, J. Ruge, T. Russel, and S. Schaffer. Hyperplane relaxation and semi-coarsening for three dimensional pde's. In *Proceedings of the Fourth Copper Mountain Conference on Multigrid Methods*, 1989.

[5] D. Gannon and J. Van Rosendale. On the impact of communication complexity in the design of parallel algorithms. *IEEE Transactions on Computers*, C-33(12):1180–1194, December 1984.

[6] D.H. Grit and J.R. McGraw. Programming divide and conquer on a multiprocessor. Technical Report UCRL-88710, Lawrence Livermore National Laboratory, 1983.

[7] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):667–677, August 1978.

[8] S.L. Johnsson. Solving tridiagonal systems on ensemble architectures. Technical Report YALEU/DCS/RR-436, Yale Research Report, November 1985.

[9] S.L. Johnsson, Y. Saad, and M.H. Schultz. Alternating direction methods on architectures. Technical Report YALEU/DCS/RR-382, Yale Research Report, October 1985.

[10] K. Stüben and U. Trottenberg. On the construction of fast solvers for elliptic equations. Technical Report IMA-Report Mr. 82.0201, GMD MBH Bonn, 1982.

[11] C. Koelbel. *Compiling programs for distributed memory machines.* PhD thesis, Purdue University, West Lafayette, IN, December 1989.

[12] C. Koelbel and P. Mehrotra. Compiler transformations for non-shared memory machines. In *International Conference on Supercomputing*, May 1989.

[13] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic process partitioning for parallel computation. *International Journal of Parallel Programming*, 16(5):366–382, 1987.

[14] D.S. Lim and R.V. Thanakij. A survey of adi implementations on hypercubes. In *Proceedings of the Second Conference on Hypercube Multiporcessors*, 1987.

[15] G.I. Marchuk. *Methods of Numerical Mathematics.* Springer-Verlag, 1975.

[16] P. Mehrotra and J. Van Rosendale. The BLAZE language: A parallel language for scientific programs. *Parallel Computing*, 5(3):339–361, November 1987.

[17] P. Mehrotra and J. Van Rosendale. Compiling high level constructs to distributed memory architectures. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[18] D.M. Nicol and J.H. Saltz. Principles for problem aggregation and assignment in medium scale multiprocessors. Technical Report 87-39, ICASE, July 1987.

[19] D.W. Peaceman and H.H. Rachford. The numerical solution of parabolic and elliptic differential equations. *SIAM Journal*, 3(1), 1955.

[20] D. Pountain. A tutorial introduction to Occam programming. Technical report, Inmos, Colorado Springs, Co., 1986.

[21] M. Rosing and R. Schnabel. An overview of Dino – a new language for numerical computation on distributed memory multiprocessors. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 312–316, 1987.

[22] G. Winter. *Fourieranalyse zur Konstruktion schneller MGR-Verfahren.* PhD thesis, Rheinischen Friedrich-Wilhelms-Universität zu Bonn, 1982.

Report Documentation Page

1. Report No
NASA CR-181900
ICASE Report No. 89-41

2. Government Accession No

3. Recipient's Catalog No

4. Title and Subtitle

PARALLEL LANGUAGE CONSTRUCTS FOR TENSOR PRODUCT COMPUTATIONS ON LOOSELY COUPLED ARCHITECTURES

5. Report Date
September 1989

6. Performing Organization Code

7. Author(s)

P. Mehrotra
J. Van Rosendale

8. Performing Organization Report No.
89-41

10. Work Unit No.
505-90-21-01

9. Performing Organization Name and Address

Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

11. Contract or Grant No.
NAS1-18605

12. Sponsoring Agency Name and Address

National Aeronautics and Space Adminsitration
Langley Research Center
Hampton, VA 23665-5225

13. Type of Report and Period Covered
Contractor Report

14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Submitted to Journal of Parallel and Distributed Computing

Final Report

16. Abstract

Distributed memory architectures offer high levels of performance and flexibility, but have proven awkard to program. Current languages for nonshared memory architectures provide a relatively low-level programming environment, and are poorly suited to modular programming, and to the construction of libraries. This paper describes a set of language primitives designed to allow the specification of parallel numerical algorithms at a higher level. We focus here on tensor product array computations, a simple but important class of numerical algorithms. We consider first the problem of programming one dimensional "kernal" routines, such as parallel tridiagonal solvers, and after that look at how such parallel kernels can be combined to form parallel tensor product algorithms.

17. Key Words (Suggested by Author(s))
parallel programming,
distributed memory architectures

18. Distribution Statement
61 - Computer Programming and Software

Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	32	A03

NASA-Langley, 1989